README for Minesweeper Game

Intro

Minesweeper is a game where mines are hidden in a grid of squares. Safe squares have numbers telling you how many mines touch the square. You can use the number clues to solve the game by opening all of the safe squares. If you click on a mine you lose the game!

Instructions

To start the game, the user only needs to run the main Python file that contains the Minesweeper UI. If the user is on macOS and has issues with Tkinter, they may need to install the standard Python distribution from python.org, which includes the correct version of Tk. Other than that, no additional setup is required.

Important Note: the "place flag" action does NOT use the usual right-click. Instead, you must use Control + Left Click to place a flag.

Features

1. Extensive use of OOP in Python, using OOPS to represent the different objects in the game (flags, bombs, etc.)
   2 Difficulty Levels:
   Beginner: 9*9, 10mines
   Intermediate: 16*16, 40mines
2. Complex Algorithm: Search algorithm to check which cells contain a mine and which do not

Justification for Complexity

This project combines multiple interacting systems, including a 2D grid, searching algorithms, and object-oriented programming. Building a Minesweeper game requires many steps and functions working together, such as revealing large empty areas without mines, checking win or loss conditions, and managing different difficulty levels. I used concepts we learnt in class, such as lists of lists, helper functions, and abstraction by breaking big tasks into functions with a single purpose each(for most of the functions). My search algorithm for revealing cells and counting nearby mines adds another layer of complexity because it must avoid re-checking cells and stay within the board boundaries. I also used variables to track game state, mines, flags, and changed cells, which all update automatically as the player interacts with the game.

Lists & Script Variables

The main list is the board, where each item represents a cell that stores several attributes such as whether it has a mine, whether it's revealed/flagged, and its adjacent-mine count. I use this 2D list to check neighbors, update cells, compute mine counts, and eventually determine whether the player has won. This list is non-trivial as it requires coordinated access using row and column indices, and many custom blocks need to read and update it.

I also used script variables to keep track of state, including the number of flags placed, whether the game is still active, and which cells changed after each click. These script variables are important for updating and showing correct graphics.

Function Table

Block / Function Name	**Domain (inputs)**	**Range (outputs)**	**Behavior (role in the context of the project)**
MinesweeperUI.__init__(self, root, rows, cols, mines)	root: Tk, rows: int, cols: int, mines: int	none	Creates the UI, initializes the board, creates button grid, sets up status label, and initializes game state display.
MinesweeperUI.create_widgets(self)	none	none	Creates all Tk widgets: grid of buttons, status label, restart/quit buttons, and binds left-click + right-click behavior. Stores all buttons in self.buttons.
MinesweeperUI.on_left_click(self, r, c)	r: int, c: int	none	Sends a reveal command to the board, updates buttons, updates status display, and shows a popup when the player wins or loses.
MinesweeperUI.on_right_click(self, r, c, event=None)	r: int, c: int, optional Tk event	none	Toggles a flag on the selected cell (if the game is still playing), refreshes the UI, and updates status counts.
MinesweeperUI.refresh_buttons(self, changed_positions)	changed_positions: List[Tuple[int,int]]	none	Updates button appearance for all affected cells: shows mine symbol, number, blank, or flag depending on cell state.
MinesweeperUI.update_status(self)	none	none	Recalculates the number of flags and updates the status label to show game state, mine count, and flag count.
MinesweeperUI.restart(sel	none	none	Starts a new game by

f)			creating a new Board object and resetting all button states.
start(difficulty_name)	difficulty_name: str matching a key in DIFFICULTIES	none	“Destroys” the difficulty menu, builds the main Minesweeper UI window with the chosen difficulty setting.
Cell.__repr__(self)	self as a Cell instance	str	Returns a readable string showing the cell’s row, column, mine status, and adjacent-mine count.
Board.__init__(self, rows, cols, mines)	rows: int, cols: int, mines: int	none	Creates a board with given dimensions, initializes all Cell objects, sets game state to PLAYING, and prepares internal grid.
Board.in_bounds(self, r, c)	r: int, c: int	bool	Checks whether a coordinate (r, c) is within the grid boundary.
Board.neighbors(self, r, c)	r: int, c: int	List[Tuple[int,int]]	Returns the list of all valid 8-direction neighbor coordinates around (r, c).
Board.plant_mines(self, first_click)	first_click: Tuple [int,int] or None	none	Randomly places mines on the board, avoiding the first-click cell and its neighbors. Also computes adjacent-mine counts for every cell.
Board.reveal(self, r, c)	r: int, c: int	List[Tuple[int,int]]	Reveals the chosen cell. Triggers mine planting on first reveal, flood-fills when adjacent count is zero, updates game state on win/loss, and returns list of all changed cell positions.
Board.toggle_flag(self, r, c)	r: int, c: int	Tuple[bool, Tuple[int,int]]	Toggles a flag on the given cell if it is not revealed. Returns whether the toggle

			succeeded and the cell’s coordinate.
Board._check_win(self)	none	bool	Checks if all non-mine cells have been revealed.
Board.get_cell(self, r, c)	r: int, c: int	Cell	Returns the Cell object at the given position.
Board.debug_print(self)	none	none	Prints a text-based representation of the board showing mines and adjacent counts (for debugging).

Video Link (DEMO):
https://drive.google.com/file/d/1a6tGWSTWtxOz_PDPP93M_8WqOd9V2u6y/view?usp=drivesdk